JAVA 8 NEW FEATURES

Table of Contents

Java 8 New Features

Before Java 8, sun people gave importance only for objects but in 1.8version oracle people gave the importance for functional aspects of programming to bring its benefits to Java.ie it doesn't mean Java is functional oriented programming language.

Java 8 is considered a landmark release because it introduced a major paradigm shift in the language and ecosystem. Here's why it's so significant:

1. Functional Programming in Java

Before Java 8, Java was purely object-oriented. Java 8 introduced:

- Lambda Expressions
- Streams API
- Functional Interfaces

This allowed developers to write declarative, functional-style code, making Java more expressive and concise.

2. Modernization of the Language

Java 8 brought features that aligned Java with modern programming trends:

- Default methods in interfaces → Easier API evolution without breaking existing implementations.
- Method References → Cleaner syntax.

3. Performance & Concurrency Enhancements

- Parallel Streams for data processing.
- CompletableFuture for asynchronous programming.
- LongAdder, StampedLock for better concurrency under high contention.

4. New Date and Time API

- Replaced the problematic java.util.Date and Calendar with immutable, thread-safe, and intuitive classes in java.time.

5. Ecosystem Impact

- Most frameworks (Spring, Hibernate, etc.) adopted Java 8 features.
- Functional programming became standard in enterprise Java.
- Improved developer productivity and readability.

6. Backward Compatibility

- Despite these big changes, Java 8 maintained backward compatibility, making migration easier for enterprises.

It was the first major update after Java 7 (2011) and set the stage for modern Java versions. Many organizations still use Java 8 because of its stability and long-term support.

Here’s a simple list of Java 8 features:

1. Lambda Expressions
2. Functional Interfaces (Predicate, Function, Consumer, Supplier, etc.)
3. Method References
4. Default and Static Methods in Interfaces
5. Streams API
6. Optional Class
7. New Date and Time API (java.time) etc..

Lambda (λ) Expression

- Lambda calculus is a big change in mathematical world which has been introduced in 1930. Because of benefits of Lambda calculus slowly these concepts started using in programming world. “LISP” is the first programming which uses Lambda Expression.
- The Main Objective of Lambda Expression is to bring benefits of functional programming into Java.

What is Lambda Expression (λ):

- Lambda Expression is just an anonymous (nameless) function. That means the function which doesn’t have the name, return type and access modifiers.
- Lambda Expression also known as anonymous functions or closures.

No-Argument Method

Original method:

```

public void m1() {
    System.out.println("hello");
}

```

Lambda equivalents:

```
() -> {
    System.out.println("hello"); //Full block syntax
}

() -> { System.out.println("hello"); } //Single statement with braces

() -> System.out.println("hello"); //Single statement without braces

```

Method with Parameters

Original method:

```
public void add(int a, int b) {
    System.out.println(a + b);
}

```

Lambda equivalent:

```
(int a, int b) -> System.out.println(a + b);

(a, b) -> System.out.println(a + b); //parameters can be ommited

```

Return Statement

Original method:

```
public String str(String str) {
    return str;
}

```

Lambda equivalent:

```
(String str) -> return str; //With explicit type and return keyword

(str) -> str; //Simplified version (type inference and no return keyword)

```

Features of Lambda Expressions

- Lambda expressions were introduced in Java 8 to enable functional programming and make code more concise and readable.
- The syntax of a lambda expression uses the arrow operator -> to separate the parameter list from the body of the expression.
    - **(parameters) -> expression or (parameters) -> { statements }.**
- Lambda expressions eliminate the need for boilerplate code such as anonymous inner classes, especially when implementing functional interfaces.
- They work only with functional interfaces, which are interfaces that contain exactly one abstract method.
- The parameter types in a lambda expression can be explicitly declared or inferred by the compiler, making the syntax shorter and cleaner.
- If the lambda body contains a **single statement,** braces {} and the return keyword can be omitted.
- Lambda expressions can capture variables from their enclosing scope, but those variables must be effectively **final**.
- They improve code readability and maintainability by allowing developers to write declarative and functional-style code.
- Lambda expressions are widely used in the **Streams API** for operations like **filtering**, **mapping**, and reducing collections.
- They support method references, which provide an even shorter syntax when calling existing methods.
- Lambda expressions can be used wherever an instance of a functional interface is expected, such as in event handling, callbacks, and concurrency tasks.

Lambda Expressions with Collections

Collection is nothing but a group of objects represented as a single entity.

The important Collection types are:

1. List(I)

2. Set(I)

3. Map(I)

Sorted Collections

Sorted List

- List(may be ArrayList, LinkedList, Vector or Stack) never talks about sorting order. If we want sorting for the list then we should use Collections class sort() method.
- Collections.sort(list)==>meant for Default Natural Sorting Order
    - For String objects: Alphabetical Order
    - For Numbers : Ascending order
- Instead of Default natural sorting order if we want customized sorting order then we should go for Comparator interface.

- Comparator interface contains only one abstract method: “compare()” Hence it is Functional interface.
- public int compare(obj1,obj2)
  - returns -ve iff obj1 must come before obj2
  - returns +ve iff obj1 must come after obj2
  - returns 0 iff obj1 and obj2 are equal
- Collections.sort(list,Comparator)==>meant for Customized Sorting Order

```java
import java.util.ArrayList;
import java.util.Collections;

class Test {
    public static void main(String[] args) {
        ArrayList<Integer> l = new ArrayList<>();
        l.add(10);
        l.add(0);
        l.add(15);
        l.add(5);
        l.add(20);

        System.out.println("Before Sorting:" + l);

        // Custom comparator that returns +1 when I1 < I2 ⇒ puts larger numbers first
        Collections.sort(l, (I1, I2) -> (I1 < I2) ? 1 : (I1 > I2) ? -1 : 0);

        System.out.println("After Sorting:" + l);
    }
}
```

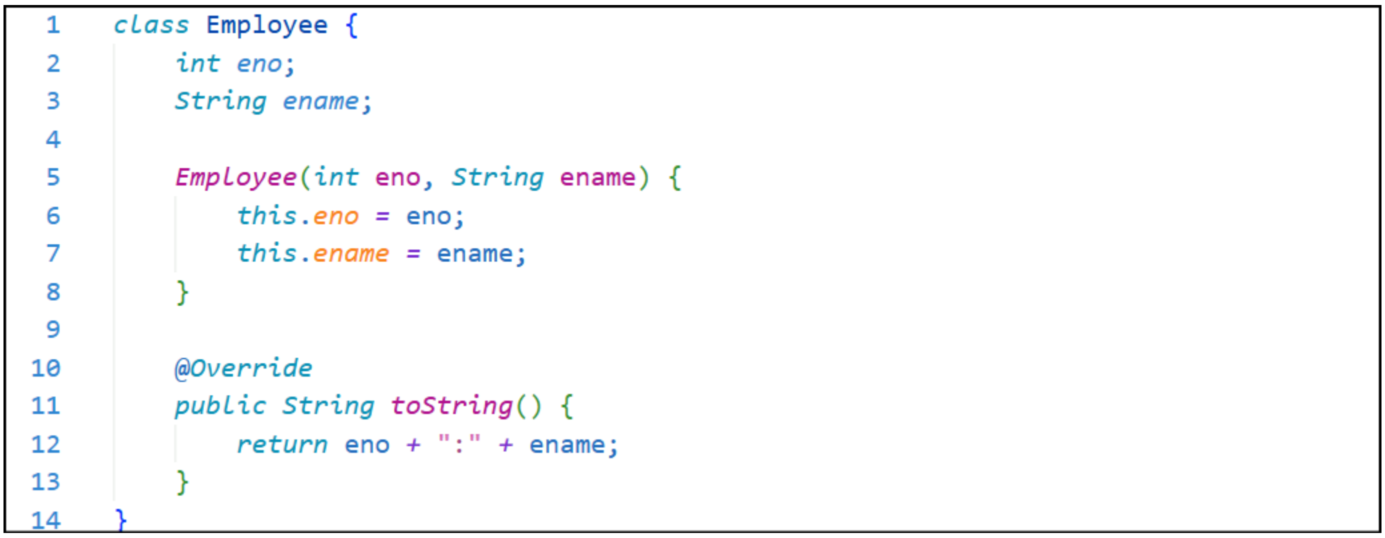
Sorted Set

- In the case of Set, if we want Sorting order then we should go for TreeSet .
- TreeSet is a sorted set backed by a **balanced Red-Black tree**; ordering is defined by either natural order (Comparable) or a supplied Comparator. Your comparator defines the reverse (descending) of natural order for Integer.
- TreeSet t = new TreeSet(); (This TreeSet object meant for default natural sorting order)
- TreeSet t = new TreeSet(Comparator c); (This TreeSet object meant for Customized Sorting Order)

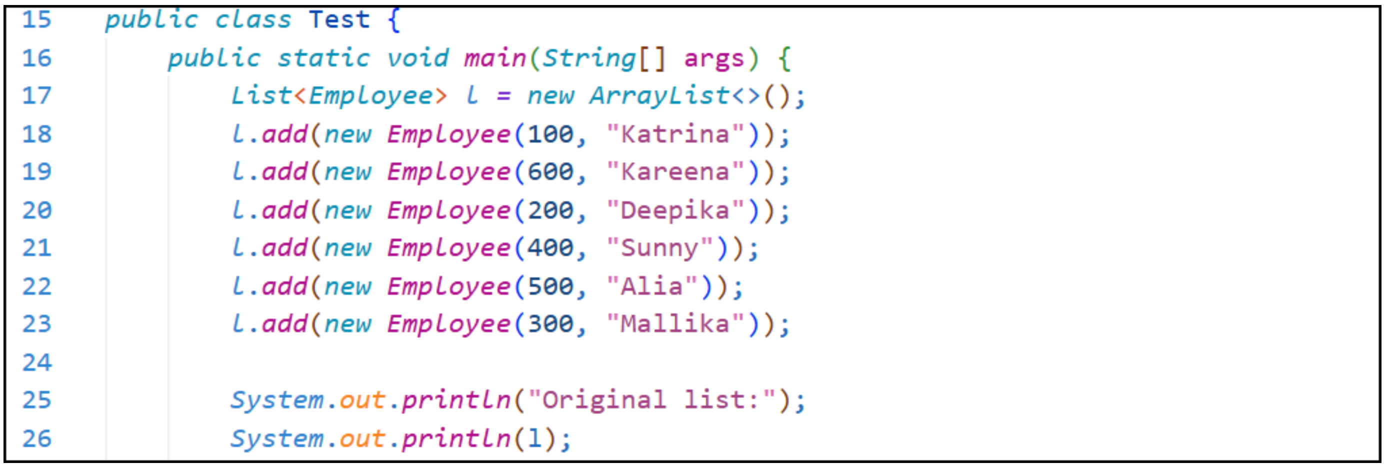
```java
public class Test {
    // Helper to add the same test data to each set
    private static void populate(TreeSet<Integer> set) {
        set.add(10);
        set.add(0);
        set.add(15);
        set.add(25);
        set.add(5);
        set.add(20);
    }
    public static void main(String[] args) {
        // -------- Original: ternary comparator (descending) --------
        TreeSet<Integer> t1 = new TreeSet<>((I1, I2) -> (I1 > I2) ? -1 : (I1 < I2) ? 1 : 0);
        populate(t1);
        System.out.println("Original (ternary comparator): " + t1); // [25, 20, 15, 10, 5, 0]

        // -------- Alternative 1: Built-in reverse order --------
        TreeSet<Integer> t2 = new TreeSet<>(Comparator.reverseOrder());
        populate(t2);
        System.out.println("Alternative 1 (Comparator.reverseOrder): " + t2); // [25, 20, 15, 10, 5, 0]

        // -------- Alternative 2: Integer.compare with reversed args --------
        TreeSet<Integer> t3 = new TreeSet<>((a, b) -> Integer.compare(b, a));
        populate(t3);
        System.out.println("Alternative 2 (Integer.compare(b, a)): " + t3); // [25, 20, 15, 10, 5, 0]
    }
}
```

Sorted Map

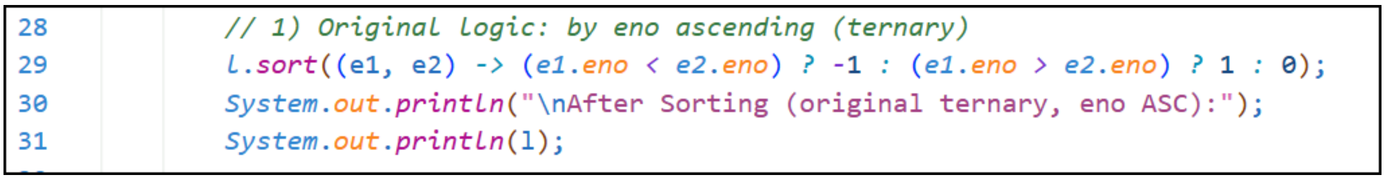
- In the case of Map, if we want default natural sorting order of keys then we should go for TreeMap.
- TreeMap m = new TreeMap(); (This TreeMap object meant for default natural sorting order of keys)
- TreeMap t = new TreeMap(Comparator c); (This TreeMap object meant for Customized Sorting Order of keys)



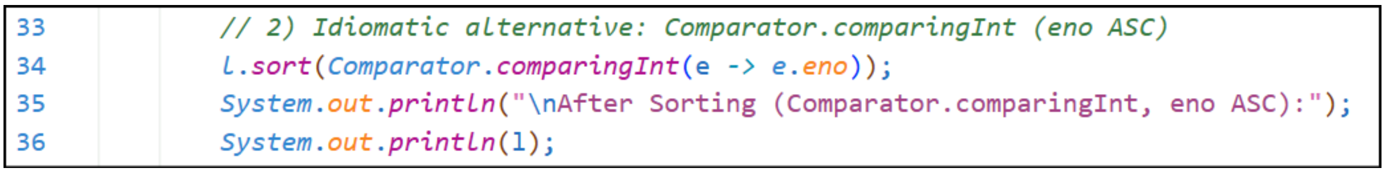
```java
class Employee {
    int eno;
    String ename;

    Employee(int eno, String ename) {
        this.eno = eno;
        this.ename = ename;
    }

    @Override
    public String toString() {
        return eno + ":" + ename;
    }
}
```



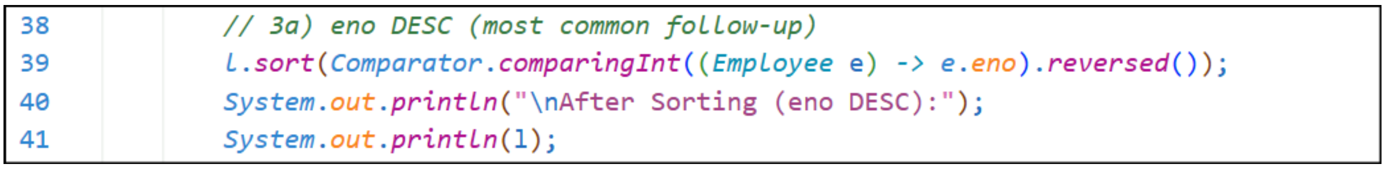
```java
public class Test {
    public static void main(String[] args) {
        List<Employee> l = new ArrayList<>();
        l.add(new Employee(100, "Katrina"));
        l.add(new Employee(600, "Kareena"));
        l.add(new Employee(200, "Deepika"));
        l.add(new Employee(400, "Sunny"));
        l.add(new Employee(500, "Alia"));
        l.add(new Employee(300, "Mallika"));

        System.out.println("Original list:");
        System.out.println(l);
```



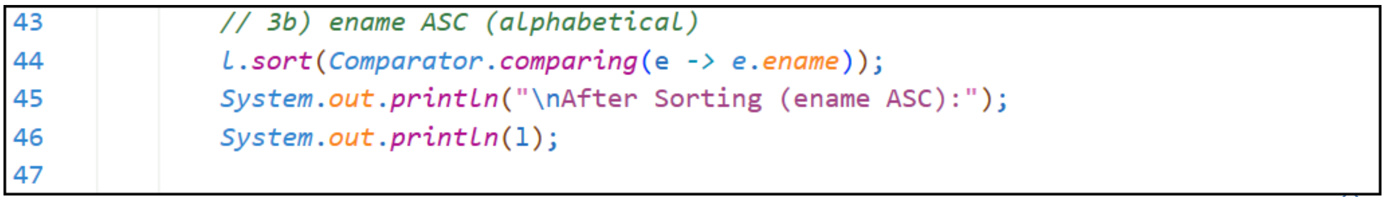
```java
        // 1) Original logic: by eno ascending (ternary)
        l.sort((e1, e2) -> (e1.eno < e2.eno) ? -1 : (e1.eno > e2.eno) ? 1 : 0);
        System.out.println("\nAfter Sorting (original ternary, eno ASC):");
        System.out.println(l);
```



```java
        // 2) Idiomatic alternative: Comparator.comparingInt (eno ASC)
        l.sort(Comparator.comparingInt(e -> e.eno));
        System.out.println("\nAfter Sorting (Comparator.comparingInt, eno ASC):");
        System.out.println(l);
```



```java
        // 3a) eno DESC (most common follow-up)
        l.sort(Comparator.comparingInt((Employee e) -> e.eno).reversed());
        System.out.println("\nAfter Sorting (eno DESC):");
        System.out.println(l);
```



```java
        // 3b) ename ASC (alphabetical)
        l.sort(Comparator.comparing(e -> e.ename));
        System.out.println("\nAfter Sorting (ename ASC):");
        System.out.println(l);

        // 3c) ename ASC, then eno ASC (stable tie-breaker)
        l.sort(Comparator.comparing((Employee e) -> e.ename)
                        .thenComparingInt(e -> e.eno));
        System.out.println("\nAfter Sorting (ename ASC, then eno ASC):");
        System.out.println(l);
    }
}
```

Why these alternatives are preferable

- **Readability & safety**: Comparator.comparingInt(e -> e.eno) expresses intent clearly and avoids mistakes in manual -1/1/0 logic.
- **Composability**: thenComparing(...) lets you chain fields (e.g., name, then id) without writing nested ternaries.
- **Direction control**: .reversed() switches ascending to descending without changing comparison logic.

Functional Interfaces

If an interface contains only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

Ex:

1. Runnable → It contains only run() method
2. Comparable → It contains only compareTo() method
3. ActionListener → It contains only actionPerformed() method
4. Callable → It contains only call() method

- Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.
- In Java 8, Sun Micro System introduced @Functional Interface annotation to specify that the interface is Functional Interface.
- Inside Functional Interface we can take only one abstract method, if we take more than one abstract method then compiler raise an error message that is called we will get compilation error.
- Inside Functional Interface we have to take exactly only one abstract method. If we are not declaring that abstract method then compiler gives an error message.

```java
@FunctionalInterface
public interface Operation<T, R> {
    // 1) Single Abstract Method (SAM)
    R apply(T input);

    // 2) Common default method: compose the next operation
    default <V> Operation<T, V> andThen(Operation<? super R, ? extends V> next) {
        return (T input) -> next.apply(this.apply(input));
    }

    // 3) Common static helper: identity transform
    static <T> Operation<T, T> identity() {
        return t -> t;
    }
}
```

Functional Interface with respect to Inheritance

- If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also Functional Interface

```java
@FunctionalInterface
interface A {
    void methodOne();
}

@FunctionalInterface
interface B extends A {
}
```

- In the child interface we can define exactly same parent interface abstract method.

```java
@FunctionalInterface
interface A {
    public void methodOne();
}

@FunctionalInterface
interface B extends A {
    public void methodOne();
}
```

- In the child interface we can't define any new abstract methods otherwise child interface won't be Functional Interface and if we are trying to use @Functional Interface annotation then compiler gives an error message.
- both parent & child interface we can write any number of default methods and there are no restrictions. Restrictions are applicable only for abstract methods.

Functional Interface Vs Lambda Expressions

- Once we write Lambda expressions to invoke its functionality, then Functional Interface is required. We can use Functional Interface reference to refer Lambda Expression.
- Wherever Functional Interface concept is applicable there we can use Lambda Expressions.

Without Lambda Expression:

```java
interface Interf {
    public void sum(int a, int b);
}

class Demo implements Interf {
    @Override
    public void sum(int a, int b) {
        System.out.println("The sum: " + (a + b));
    }
}

public class Test {
    public static void main(String[] args) {
        Interf i = new Demo();
        i.sum(20, 5); // Output: The sum: 25
    }
}
```

With Lambda expression:

```java
@FunctionalInterface
interface Interf {
    void sum(int a, int b);
}

public class Test {
    public static void main(String[] args) {
        // Lambda implementation instead of a concrete class
        Interf i = (a, b) -> System.out.println("The sum: " + (a + b));
        i.sum(20, 5); // Output: The sum: 25
    }
}
```

Anonymous inner classes vs Lambda Expressions

Wherever we are using anonymous inner classes there may be a chance of using Lambda expression to reduce length of the code and to resolve complexity.

With anonymous inner class

```java
class Test {
    public static void main(String[] args) {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    System.out.println("Child Thread");
                }
            }
        });

        t.start();

        for (int i = 0; i < 10; i++) {
            System.out.println("Main Thread");
        }
    }
}
```

With Lambda expression

```java
class Test {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.println("Child Thread");
            }
        });

        t.start();

        for (int i = 0; i < 10; i++) {
            System.out.println("Main Thread");
        }
    }
}
```

What are the advantages of Lambda expression

- ❖ We can reduce length of the code so that readability of the code will be improved.
- ❖ We can resolve complexity of anonymous inner classes.
- ❖ We can provide Lambda expression in the place of object.
- ❖ We can pass lambda expression as argument to methods.

Note:

- ❖ Anonymous inner class can extend concrete class, can extend abstract class, can implement interface with any number of methods but
- ❖ Lambda expression can implement an interface with only single abstract method (Functional Interface).
- ❖ Hence if anonymous inner class implements Functional Interface in that case only we can replace with lambda expressions. Hence wherever anonymous inner class concept is there, it may not be possible to replace with Lambda expressions.
- ❖ Anonymous inner class! = Lambda Expression

- Inside anonymous inner class we can declare instance variables.
- Inside anonymous inner class “this” always refers current inner class object(anonymous inner class) but not related outer class object
- Inside lambda expression we can’t declare instance variables.
- Whatever the variables declare inside lambda expression are simply acts as local variables
- Within lambda expression ‘this” keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression)
- From lambda expression we can access enclosing class variables and enclosing method variables directly.
- The local variables referenced from lambda expression are implicitly final and hence we can’t perform re-assignment for those local variables otherwise we get compile time error.

Differences between anonymous inner classes and Lambda expression

Aspect	**Anonymous Inner Class**	**Lambda Expression**
Definition	It’s a class without a name	It’s a method without a name (anonymous function)
Inheritance	Can extend abstract and concrete classes	Cannot extend abstract and concrete classes
Interface Implementation	Can implement an interface with any number of abstract methods	Can implement an interface with a single abstract method (Functional Interface)
Instance Variables	Can declare instance variables inside the class	Cannot declare instance variables; variables act as local variables
Instantiation	Can be instantiated	Cannot be instantiated
this Keyword	Refers to the current anonymous inner class object	Refers to the enclosing outer class object
Best Use Case	Best choice for handling multiple methods	Best choice for handling an interface with a single abstract method
Compilation	Generates a separate .class file (e.g., OuterClass$1.class)	No separate .class file; converted into a private method in the outer class
Memory Allocation	Allocated on demand; resides in permanent memory of JVM (Method Area)	No permanent allocation; treated as a method reference

Predicates(I)

- A predicate is a function with a single argument and returns boolean value.
- To implement predicate functions in Java, Oracle people introduced Predicate interface in 1.8 version (Predicate<T>).

- Predicate interface present in Java.util.function package.
- It’s a functional interface and it contains only one method i.e., test()
- As predicate is a functional interface and hence it can refer lambda expression

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Write a predicate to check whether the given integer is greater than 10 or not.

```java
import java.util.function.Predicate;

public class PredicateExample {
    public static void main(String[] args) {
        // Predicate to check if number is greater than 10
        Predicate<Integer> isGreaterThan10 = n -> n > 10;

        // Test the predicate
        System.out.println(isGreaterThan10.test(15)); // true
        System.out.println(isGreaterThan10.test(8));  // false
    }
```

Predicate joining

- It’s possible to join predicates into a single predicate by using the following methods.
    - and()
    - or()
    - negate()
- these are exactly same as logical AND ,OR complement operators

```java
import java.util.function.Predicate;

public class Test {

    public static void m1(Predicate<Integer> p, int[] x) {
        for (int x1 : x) {
            if (p.test(x1)) {
                System.out.println(x1);
            }
        }
    }
```

```java
    public static void main(String[] args) {
        int[] x = {0, 5, 10, 15, 20, 25, 30};

        Predicate<Integer> p1 = i -> i > 10;
        Predicate<Integer> p2 = i -> i % 2 == 0;

        System.out.println("The Numbers Greater Than 10:");
        m1(p1, x);

        System.out.println("The Even Numbers Are:");
        m1(p2, x);

        System.out.println("The Numbers Not Greater Than 10:");
        m1(p1.negate(), x);

        System.out.println("The Numbers Greater Than 10 And Even Are:");
        m1(p1.and(p2), x);

        System.out.println("The Numbers Greater Than 10 OR Even:");
        m1(p1.or(p2), x);
    }
}
```

Output:

```
The Numbers Greater Than 10:
15
20
25
30
The Even Numbers Are:
0
10
20
30
The Numbers Not Greater Than 10:
0
5
10
The Numbers Greater Than 10 And Even Are:
20
30
The Numbers Greater Than 10 OR Even:
0
10
15
20
25
30
```

Functions(I)

- Functions are exactly same as predicates except that functions can return any type of result but function should (can) return only one value and that value can be any type as per our requirement.
- To implement functions oracle people introduced Function interface in 1.8version.
- Function interface present in Java.util.function package.
- Functional interface contains only one method i.e., apply()

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

- This represents a generic functional interface with:
    - T → Input type
    - R → Return type

Write a function to find length of given input string.

```java
import java.util.function.Function;

public class StringLengthExample {
    public static void main(String[] args) {
        // Function to find length of a string
        Function<String, Integer> findLength = str -> str.length();

        // Test the function
        String input = "Hello Java!";
        System.out.println("Length of \"" + input + "\" is: " + findLength.apply(input));
    }

```

Differences between predicate and function

Predicate	**Function**
To implement conditional checks we should go for predicate	To perform certain operation and to return some result we should go for function
Predicate can take one type parameter which represents input argument type. Predicate<T>	Function can take 2 type parameters. First one represents input argument type and second one represents return type. Function<T,R>
Predicate interface defines only one method called test()	Function interface defines only one method called apply()
public boolean test(T t)	public R apply(T t)
Predicate can return only boolean value	Function can return any type of value

Consumer (I)

❖ Sometimes our requirement is we must provide some input value, perform certain operation, but not require returning anything, then we should go for Consumer. I.e Consumer can be used to consume object and perform certain operation.

```java
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

```

❖ **Type Parameter T**: The type of the input argument.
❖ **Method accept(T t)**: Performs an operation on the given argument and returns **no result**.
❖ It's commonly used in **lambda expressions** and **method references**.

Example:

```java
import java.util.function.*;
import java.util.*;

class Movie {
    String name;
    String hero;
    String heroine;

    Movie(String name, String hero, String heroine) {
        this.name = name;
        this.hero = hero;
        this.heroine = heroine;
    }
}
```

```java
class Test {
    public static void main(String[] args) {
        ArrayList<Movie> l = new ArrayList<>();
        populate(l);

        Consumer<Movie> c = m -> {
            System.out.println("Movie Name: " + m.name);
            System.out.println("Movie Hero: " + m.hero);
            System.out.println("Movie Heroine: " + m.heroine);
            System.out.println();
        };

        for (Movie m : l) {
            c.accept(m);
        }
    }

    public static void populate(ArrayList<Movie> l) {
        l.add(new Movie("Bahubali", "Prabhas", "Anushka"));
        l.add(new Movie("Raees", "Shah Rukh", "Mahira"));
        l.add(new Movie("Dangal", "Aamir", "Sakshi"));
        l.add(new Movie("Sultan", "Salman", "Anushka"));
    }
}
```

Consumer Chaining

- ❖ Just like Predicate Chaining and Function Chaining, Consumer Chaining is also possible. For this Consumer Functional Interface contains default method andThen().
- ❖ c1.andThen(c2).andThen(c3).accept(s) First Consumer c1 will be applied followed by c2 and c3.

Example:

```java
import java.util.function.Consumer;

class Movie {
    String name, hero, heroine;
    Movie(String name, String hero, String heroine) {
        this.name = name;
        this.hero = hero;
        this.heroine = heroine;
    }
}

```

```java
public class Test {
    public static void main(String[] args) {
        Movie m = new Movie("Bahubali", "Prabhas", "Anushka");

        Consumer<Movie> printName    = x -> System.out.println("Name: " + x.name);
        Consumer<Movie> printHero    = x -> System.out.println("Hero: " + x.hero);
        Consumer<Movie> printHeroine = x -> System.out.println("Heroine: " + x.heroine);
        Consumer<Movie> newline      = x -> System.out.println();

        Consumer<Movie> printAll = printName
                .andThen(printHero)
                .andThen(printHeroine)
                .andThen(newline);

        printAll.accept(m); // Executes in the defined order //Name: Bahubali Hero: Prabhas Heroine: Anushka
    }
}
```

Supplier (I)

- A Supplier is a functional interface that takes no input and returns a value of type T.
    - supply Student object
    - Supply Random Name
    - Supply Random OTP
    - Supply Random Password etc For this type of requirements we should go for Supplier.
- Supplier can be used to supply items (objects).
- Supplier won't take any input and it will always supply objects.
- Supplier Functional Interface contains only one method get().

```java
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Example:

```java
import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<Double> randomValue = () -> Math.random();
        System.out.println("Random Value: " + randomValue.get());
    }
```

Comparison Table of Predicate, Function, Consumer, and Supplier

Property	Predicate	Function	Consumer	Supplier
1) Purpose	To take some input and perform conditional checks	To take some input and perform required operation and return the result	To consume some input and perform required operation (no return)	To supply some value based on requirement
2) Interface Declaration	interface Predicate<T> { … }	interface Function<T, R> { … }	interface Consumer<T> { … }	interface Supplier<R> { … }
3) Single Abstract Method (SAM)	boolean test(T t)	R apply(T t)	void accept(T t)	R get()
4) Default Methods	and(), or(), negate()	andThen(), compose()	andThen()	-
5) Static Method	isEqual()	identity()	-	-

Two-Argument (Bi) Functional Interfaces

Normal Functional Interfaces (Predicate, Function and Consumer) can accept only one input argument. But sometimes our programming requirement is to accept two input arguments, then we should go for two-argument functional interfaces. The following functional interfaces can take 2 input arguments.

1. BiPredicate
2. BiFunction
3. BiConsumer

BiPredicate(I)

- ❖ Normal Predicate can take only one input argument and perform some conditional check. Sometimes our programming requirement is we have to take 2 input arguments and perform some conditional check, for this requirement we should go for BiPredicate.
- ❖ BiPredicate is exactly same as Predicate except that it will take 2 input arguments.

```java
@FunctionalInterface
public interface BiPredicate<T1, T2> {
    boolean test(T1 t1, T2 t2);
```

Example:

```java
import java.util.function.BiPredicate;

class Test {
    public static void main(String[] args) {
        // BiPredicate to check if sum of two integers is even
        BiPredicate<Integer, Integer> p = (a, b) -> (a + b) % 2 == 0;

        System.out.println(p.test(10, 20)); // true (30 is even)
        System.out.println(p.test(15, 20)); // false (35 is odd)
    }
}
```

BiFunction(I)

- ❖ Normal Function can take only one input argument and perform required operation and returns the result. The result need not be boolean type.
- ❖ But sometimes our programming requirement to accept 2 input values and perform required operation and should return the result. Then we should go for BiFunction.
- ❖ BiFunction is exactly same as function except that it will take 2 input arguments.

```java
interface BiFunction<T, U, R> {
    public R apply(T t, U u);
    // default method andThen()
```

Example:

```java
import java.util.function.*;

class Test {
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> f = (a, b) -> a * b;
        System.out.println(f.apply(10, 20));    // Output: 200
        System.out.println(f.apply(100, 200));  // Output: 20000
    }
}
```

BiConsumer(I)

- Normal Consumer can take only one input argument and perform required operation and won't return any result.
- But sometimes our programming requirement to accept 2 input values and perform required operation and not required to return any result. Then we should go for BiConsumer.
- BiConsumer is exactly same as Consumer except that it will take 2 input arguments.

```java
@FunctionalInterface
interface BiConsumer<T, U> {
    void accept(T t, U u);
}

```

Example:

```java
import java.util.function.*;

class Test {
    public static void main(String[] args) {
        BiConsumer<String, String> c = (s1, s2) -> System.out.println(s1 + s2);
        c.accept("durga", "soft");  // Output: durgasoft
    }
}
```

Default Methods

- Until 1.7 version onwards inside interface we can take only public abstract methods and public static final variables (every method present inside interface is always public and abstract whether we are declaring or not).
- Every variable declared inside interface is always public static final whether we are declaring or not.
- But from 1.8 version onwards in addition to these, we can declare default concrete methods also inside interface, which are also known as defender methods.
- We can declare default method with the keyword “default” as follows

```java
interface MyInterface {
    // Default method
    default void m1() {
        System.out.println("Default Method");
    }
}

public class DefaultMethodDemo implements MyInterface {
    public static void main(String[] args) {
        DefaultMethodDemo demo = new DefaultMethodDemo();
        demo.m1(); // Calling the default method
    }
}
```

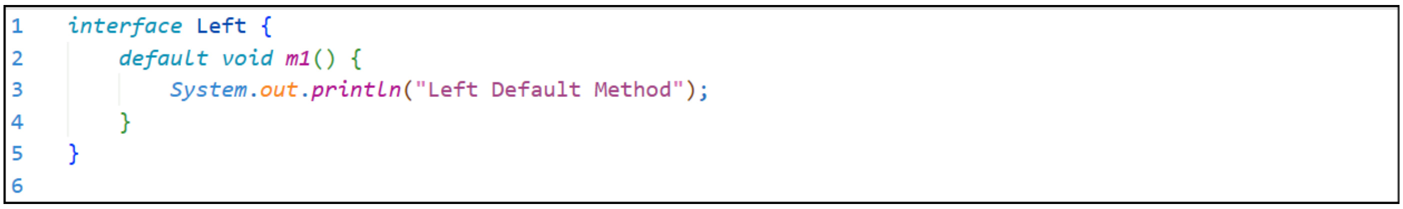
- Interface default methods are by-default available to all implementation classes. Based on requirement implementation class can use these default methods directly or can override.
- Default methods also known as defender methods or virtual extension methods.
- The main advantage of default methods is without effecting implementation classes we can add new functionality to the interface (backward compatibility).

If a new version of a software or platform is backward compatible, it means Programs, files, or features created in older versions will still run or work correctly in the new version. Users don't need to change their existing code or data when upgrading.

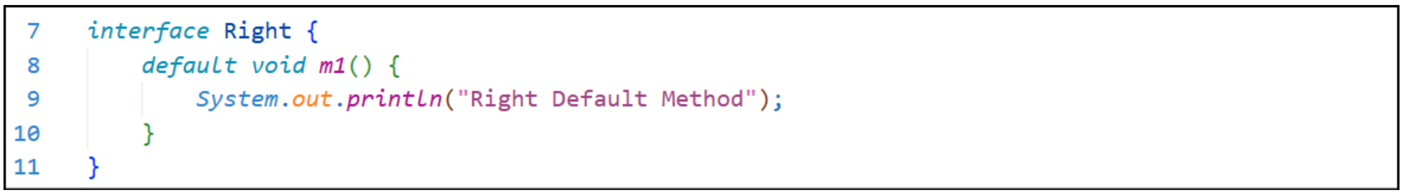
Default method vs multiple inheritance:

Two interfaces can contain default method with same signature then there may be a chance of ambiguity problem (diamond problem) to the implementation class. To overcome this problem compulsory we should override default method in the implementation class otherwise we get compile time error.

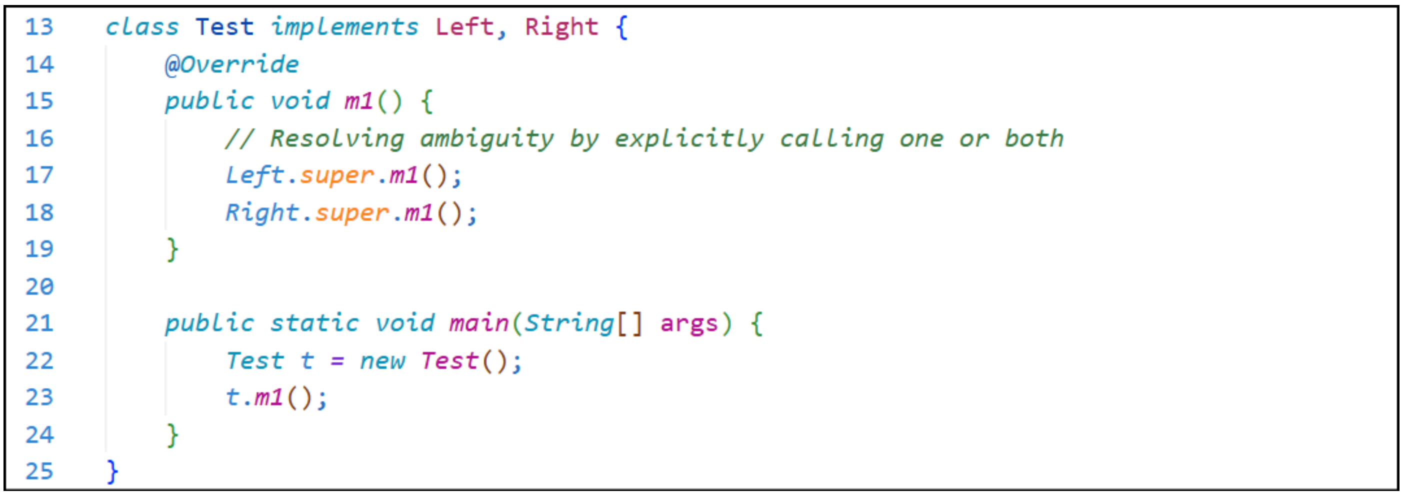
In the implementation class we can provide completely new implementation or we can call any interface method as follows. interfacename.super.m1();



```java
interface Left {
    default void m1() {
        System.out.println("Left Default Method");
    }
}

```



```java
interface Right {
    default void m1() {
        System.out.println("Right Default Method");
    }
}
```



```java
class Test implements Left, Right {
    @Override
    public void m1() {
        // Resolving ambiguity by explicitly calling one or both
        Left.super.m1();
        Right.super.m1();
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.m1();
    }
}
```

Differences between interface with default methods and abstract class

Even though we can add concrete methods in the form of default methods to the interface, it won't be equal to abstract class.

Interface with Default Methods	Abstract Class
Inside interface every variable is always public static final and there is no chance of instance variables	Inside abstract class there may be a chance of instance variables which are required to the child class
Interface never talks about state of object	Abstract class can talk about state of object

Interface with Default Methods	Abstract Class
Inside interface we can't declare constructors	Inside abstract class we can declare constructors
Inside interface we can't declare instance and static blocks	Inside abstract class we can declare instance and static blocks
Functional interface with default methods can refer lambda expression	Abstract class can't refer lambda expressions
Inside interface we can't override Object class methods	Inside abstract class we can override Object class methods

Static methods

- From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions.
- Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can't call interface static methods. We should call interface static methods by using interface name.
- As interface static methods by default not available to the implementation class, overriding concept is not applicable.
- Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

```java
interface Interf {
    public static void sum(int a, int b) {
        System.out.println("The Sum: " + (a + b));
    }
}

class Test implements Interf {
    public static void main(String[] args) {
        Test t = new Test();
        // t.sum(10, 20); // CE (Compile-time Error)
        // Test.sum(10, 20); // CE (Compile-time Error)
        Interf.sum(10, 20); // Correct way to call static method in interface
    }
}
```

- From 1.8 version onwards we can write main() method inside interface and hence we can run interface directly from the command prompt.

```java
interface Interf {
    public static void main(String[] args) {
        System.out.println("Interface Main Method");
    }
}

```

Method and Constructor references by using (::)

- ❖ Functional Interface method can be mapped to our specified method by using :: (double colon) operator. This is called method reference.
- ❖ Our specified method can be either static method or instance method.
- ❖ Functional Interface method and our specified method should have same argument types, except this the remaining things like
- ❖ returntype, methodname, modifiersetc are not required to match.
- ❖ Functional Interface can refer lambda expression and Functional Interface can also refer method reference. Hence lambda expression can be replaced with method reference. Hence method reference is alternative syntax to lambda expression.

If the method you want to refer to is **static**

```
ClassName::methodName
```

If the method is **instance-level**

```
objectRef::methodName
```

Example:

```java
interface Interf {
    void m1(int i);
}

class Test {
    public void m2(int i) {
        System.out.println("From Method Reference: " + i);
    }

    public static void main(String[] args) {
        // Lambda expression implementing Interf.m1(int)
        Interf f = i -> System.out.println("From Lambda Expression: " + i);
        f.m1(10);

        // Instance method reference: objectRef::methodName
        Test t = new Test();
        Interf i1 = t::m2;
        i1.m1(20);
    }
}
```

- ❖ We can use :: (double colon)operator to refer constructors also

```java
class Sample {
    private String s;
    Sample(String s) {
        this.s = s;
        System.out.println("Constructor Executed:" + s);
    }
}

interface Interf {
    Sample get(String s);
}

class Test {
    public static void main(String[] args) {
        Interf f  = s -> new Sample(s); // Lambda calling the constructor
        f.get("From Lambda Expression");

        Interf f1 = Sample::new;        // constructor reference
        f1.get("From Constructor Reference");
    }
}
```

Streams

- ❖ Java Streams (introduced in Java 8) provide a functional, declarative way to process data from collections, arrays, or other sources. Instead of writing loops, you build pipelines of operations:
  - ❖ Source → Where data comes from (e.g., List, Set, array).
  - ❖ Intermediate Operations → Transformations like filter(), map(), sorted().
  - ❖ Terminal Operations → Produce a result or side effect like collect(), forEach(), reduce().
- ❖ Streams do not store data; they process elements on demand and support lazy evaluation and parallel execution.

```
Stream s = c.stream();
```

Here, c represents a collection (like List, Set, etc.), and stream() is called on it to create a Stream.

- ❖ Stream is an **interface** in java.util.stream.
- ❖ Once you obtain a stream from a collection, you can process its elements using the Stream API.
- ❖ Processing occurs in **two phases**:
  1. **Configuration** – Setting up the pipeline with intermediate operations (e.g., filter, map, sorted).
  2. **Processing** – Triggering execution with a terminal operation (e.g., collect, forEach, reduce).

1.Configuration Phase

- ❖ This phase is about **setting up the stream pipeline** using **intermediate operations**. These operations do not execute immediately; they are **lazy** and only run when a terminal operation is called.
- ❖ **Key Characteristics**:
  - o **Intermediate operations** like filter() and map() are part of the configuration phase.
  - o They **do not execute immediately**; they build a pipeline.
  - o Execution happens only when a **terminal operation** (like collect() or forEach()) is called.

Filtering

Purpose: We can configure a filter to filter elements from the collection based on some **boolean condition** by using filter()method of Stream interface.

Method:

```
public Stream<T> filter(Predicate<T> t)
```

Predicate<T> is a functional interface that represents a condition returning true or false, often implemented using **lambda expressions**.

Example:

```
Stream s = c.stream();                    // Create stream from collection
Stream s1 = s.filter(i -> i % 2 == 0);    // Filter even numbers
```

Key Point: Use filter() when you need to **exclude elements** that do not satisfy a condition.

Mapping

Purpose: If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

Method:

```java
public Stream<R> map(Function<T, R> f)
```

Function<T, R> takes an input and returns a transformed output, it can also be a lambda expression.

Example:

```java
Stream s = c.stream();                    // Create stream
Stream s1 = s.map(i -> i + 10);           // Add 10 to each element
```

Key Point: Use map() when you want to **change the data** (e.g., apply a function to each element).

2.Processing Phase

- This is where the configured stream pipeline is executed using **terminal operations**.
- **Key Characteristics**:
  - **Triggers execution** of all intermediate operations (which are lazy).
  - Produces a **result** or **side effect** (e.g., printing, collecting).
  - After processing, the stream is **consumed** and cannot be reused.

Processing by collect() method

The collect() method in Java Streams is a terminal operation used during the Processing phase. It gathers the elements of a stream into a container such as a List, Set, or even a custom structure.

Approach-1: Without Streams

```java
class Test {
    public static void main(String[] args) {
        ArrayList<Integer> l1 = new ArrayList<>();
        for (int i = 0; i <= 10; i++) {
            l1.add(i);
        }
        System.out.println("Original List: " + l1);

        ArrayList<Integer> l2 = new ArrayList<>();
        for (Integer i : l1) {
            if (i % 2 == 0) {
                l2.add(i);
            }
        }
        System.out.println("Even Numbers: " + l2);
    }
}
```

Approach-2: With Streams

```java
class Test {
    public static void main(String[] args) {
        ArrayList<Integer> l1 = new ArrayList<>();
        for (int i = 0; i <= 10; i++) {
            l1.add(i);
        }
        System.out.println("Original List: " + l1);

        ArrayList<Integer> l2 = new ArrayList<>();
        for (Integer i : l1) {
            if (i % 2 == 0) {
                l2.add(i);
            }
        }
        System.out.println("Even Numbers: " + l2);
    }
}
```

Program for map() and collect() Method

```java
class Test {
    public static void main(String[] args) {
        // Create and populate the list
        ArrayList<String> l1 = new ArrayList<>();
        l1.add("rvk");
        l1.add("rvk");
        l1.add("rk");
        l1.add("rkv");
        l1.add("rvki");
        l1.add("rvkir");

        System.out.println("Original List: " + l1);

        // Use Stream to convert each string to uppercase and collect into a new list
        List<String> l2 = l1.stream()
                            .map(s -> s.toUpperCase())
                            .collect(Collectors.toList());

        System.out.println("Uppercase List: " + l2);
    }
}
```

Processing by the count() Method

- ❖ The count() method is defined in java.util.stream.Stream.
- ❖ It returns the total number of elements present in the stream as a long value.
- ❖ Invoking this method triggers execution of the entire pipeline, including any previously specified lazy operations.

```java
import java.util.*;
import java.util.stream.*;

class Test {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

        long evenCount = numbers.stream()
                                .filter(i -> i % 2 == 0) // configuration
                                .count();                // processing

        System.out.println("Count of even numbers: " + evenCount);
    }
}
```

Processing by sorted()method

- ❖ If we sort the elements present inside stream then we should go for sorted() method.
- ❖ The sorting can either default natural sorting order or customized sorting order specified by comparator.

```java
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
```

Example:

```java
import java.util.*;
import java.util.stream.*;

class Test {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 3);

        List<Integer> sortedList = numbers.stream()
                                          .sorted() // natural order
                                          .collect(Collectors.toList());

        System.out.println("Sorted List: " + sortedList);
    }
}

```

Sort strings by length:

```java
List<String> names = Arrays.asList("Java", "Streams", "API");

List<String> sortedByLength = names.stream()
                                   .sorted(Comparator.comparingInt(String::length))
                                   .collect(Collectors.toList());

System.out.println(sortedByLength);
```

Processing by min() and max() methods

❖ The min() and max() methods in Java Streams are terminal operations used during the Processing phase to find the smallest or largest element in a stream based on natural ordering or a custom comparator.

```java
Optional<T> min(Comparator<? super T> comparator)
Optional<T> max(Comparator<? super T> comparator)
```

❖ Both return an Optional because the stream might be empty.

❖ Require a Comparator to define the ordering.

Example:

```java
import java.util.*;
import java.util.stream.*;

class Test {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 3);

        Optional<Integer> minValue = numbers.stream()
                                            .min(Integer::compareTo);

        Optional<Integer> maxValue = numbers.stream()
                                            .max(Integer::compareTo);

        System.out.println("Min: " + minValue.get());
        System.out.println("Max: " + maxValue.get());
    }
}
```

forEach() method

- The forEach() method in Java Streams is a terminal operation that, during processing, goes through each element in the stream to perform a specified action.
- It is defined in java.util.stream.Stream and is used to apply an action—often a lambda expression or method reference—to every item in the stream.

```
void forEach(Consumer<? super T> action)
```

- Using forEach() will execute all intermediate operations in the pipeline. Since it triggers side effects, forEach() returns void and does not create a new stream.

```
import java.util.*;
import java.util.stream.*;

class Test {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Java", "Streams", "API");

        names.stream()
             .forEach(System.out::println); // Method reference
    }
}
```

toArray() method

- In Java Streams, the toArray() method is a terminal operation used to turn the stream's elements into an array. This comes in handy when you need an array rather than a List or another collection after your processing.

```
Object[] toArray()
<A> A[] toArray(IntFunction<A[]> generator)
```

- The first version of toArray() returns an Object[]. The second version lets you specify the type of array by providing a generator, such as String[]::new.

```
import java.util.*;
import java.util.stream.*;

class Test {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Using toArray() without type
        Object[] objArray = numbers.stream().toArray();
        System.out.println(Arrays.toString(objArray));

        // Using toArray() with type
        Integer[] intArray = numbers.stream().toArray(Integer[]::new);
        System.out.println(Arrays.toString(intArray));
    }
}
```

Stream.of()method

- The Stream.of() method in Java allows you to easily build a stream from a set list of elements. Introduced with Java 8's Stream API, it's found in java.util.stream.Stream.
- Its main function is to produce a sequential stream using the provided values, returning a Stream<T> that holds those elements.

```
static <T> Stream<T> of(T... values)
```

Example:

```
import java.util.stream.*;

class Test {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("Java", "Streams", "API");
        stream.forEach(System.out::println);
    }
}
```

Date and Time API: (Joda-Time API)

- Until Java 1.7version the classes present in Java.util package to handle Date and Time (like Date, Calendar, TimeZone etc) are not up to the mark with respect to convenience and performance.
- To overcome this problem in the 1.8version oracle people introduced Joda-Time API. This API developed by joda.org and available in Java in the form of Java.time package.

```
import java.time.LocalDateTime;

public class Test {
    public static void main(String[] args) {
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("Year: " + dt.getYear());
        System.out.println("Month: " + dt.getMonth());
        System.out.println("Day: " + dt.getDayOfMonth());
        System.out.println("Hour: " + dt.getHour());
        System.out.println("Minute: " + dt.getMinute());
        System.out.println("Current Date and Time: " + dateTime);
    }
}
```

To Represent Zone

- ZoneId object can be used to represent Zone.

```
ZoneId zone = ZoneId.of("America/New_York");
System.out.println(zone);
```

- We can create ZoneId for a particular zone as follows

```
ZoneId defaultZone = ZoneId.systemDefault();
System.out.println("Default Zone: " + defaultZone);

Set<String> zones = ZoneId.getAvailableZoneIds();
System.out.println("Total Zones: " + zones.size());
```

Period Object

In Java's **Date-Time API (java.time)**, a **Period** object represents a **quantity of time in terms of years, months, and days**. It is used for **date-based calculations**, not time-of-day or time zones.

```java
LocalDate birthDate = LocalDate.of(1995, 8, 20);
LocalDate today = LocalDate.now();
Period age = Period.between(birthDate, today);
System.out.println("Age: " + age.getYears() + " years, " +
                   age.getMonths() + " months, " +
                   age.getDays() + " days");
```